Verification of Timed Circuits with Failure Directed Abstractions *

Hao Zheng,† Chris J. Myers, David Walter, Scott Little, and Tomohiro Yoneda ‡
University of Utah
Salt Lake City, UT 84112
{hao,myers,dwalter,little}@vlsigroup.ece.utah.edu, yoneda@nii.ac.jp

Abstract

This paper presents a method to address state explosion in timed circuit verification by using abstraction directed by the failure model. This method allows us to decompose the verification problem into a set of subproblems, each of which proves that a specific failure condition does not occur. To each subproblem, abstraction is applied using safe transformations to reduce the complexity of verification. The abstraction preserves all essential behaviors conservatively for the specific failure model in the concrete description. Therefore, no violations of the given failure model are missed when only the abstract description is analyzed. An algorithm is also shown to examine the abstract error trace to either find a concrete error trace or report that it is a false negative. This paper presents results using the proposed failure directed abstractions as applied to two large timed circuit designs.

1. Introduction

Timed circuits are defined to be any circuits that are aggressively optimized using timing assumptions such that their correctness is dependent on these assumptions. Utilizing timing assumptions can produce circuits with a significant improvement in speed as demonstrated by their use in a gigahertz research microprocessor (guTS) at IBM [12] and by the RAPPID instruction length decoder designed at Intel [24]. The correctness of these new timed circuit styles is highly dependent upon their timing assumptions. Therefore, extensive timing verification is necessary during the design process.

State explosion is a serious challenge for state space exploration based verification approaches. Many methods exist to address the state explosion problem. *Symbolic model checking* [4] represents the state space implicitly using *Binary Decision Diagrams* (BDDs), and is able to handle systems with substantially increased sizes. Applying decision diagrams to timing verification has also been successful [3, 20, 16]. Since interleavings among the concurrent events are the main source of state explosion, a number of techniques have been proposed to reduce the number of interleavings to be explored using *partial orders* [27, 8]. There has also been some success in adapting these methods to timing verification [2, 29]. While both BDDs and partial orders allow the verification of larger systems, many practical timed circuits are still too large to be efficiently analyzed using these techniques alone.

Compositional reasoning and *abstraction* are essential to verifying large systems. Compositional verification based on *assume-guarantee* style reasoning explores the inherent modular structure in systems [19, 14, 9, 11, 17], and it has been applied to the verification of timed circuits [26]. Compositional verification makes assumptions about the environment with which the system interacts, then checks these assumptions later. These assumptions are typically generated by hand. Therefore, if the system has complex interactions with its environment, it can be difficult to make accurate assumptions. Abstraction produces the reduced model of a system by abstracting away certain details that are unnecessary when reasoning about the system [5, 6]. In [1], hand abstractions are used for the verification of timed synchronous domino circuits in the guTS design [12]. In both cases, the assumptions and abstractions are generated by hand, making these techniques difficult to apply except by an expert user. In [13], a hierarchical approach similar to that in [7] is presented. In this approach, an abstraction for each module in a system is found and verification is applied to the composition of those abstractions. In [15], a constraint oriented proof methodology is applied to verify infinite systems. Constraints on infinite systems are broken into an infinite number of simple constraints on finite systems, then these constraints are grouped into finite equivalent classes. However, this methodology is not complete

*This research is supported by SRC contract 2002-TJ-1024, NSF Japan Program award INT-0087281, and JSPS Joint Research Projects.

†Hao Zheng is with IBM Microelectronics in Burlington, VT.

‡Tomohiro Yoneda is with the National Institute of Informatics in Tokyo, Japan.

in that the reduction of infinite systems is not guaranteed. In [10], a software model checking method utilizing *lazy abstraction* is presented to improve performance by adding information during abstraction refinement only when necessary. It would be interesting to see if this method can be adapted to hardware verification.

A method that combines compositional reasoning and abstraction to reduce the cost of timing verification is presented in [30]. By utilizing the inherent modular structure in hardware designs, each module in a design is verified individually. Before verification, information in the environment that is irrelevant to reasoning about the module being verified is abstracted away. Then, that module is verified with its abstracted environment. While this work has been shown to verify larger circuits, it cannot be applied to flat designs or ones where the size of individual modules is beyond the capacity of the timing verification tool. In these cases, the module must first be decomposed by hand into smaller submodules

This paper addresses this problem by dividing the verification problem as directed by the failure model rather than the module interface boundaries. Timing verification is utilized to show that several different failure conditions cannot arise. This paper proposes to decompose the verification problem into several subproblems in which each of the failure conditions is checked individually. In this form of problem decomposition, any information in a model irrelevant to a given failure condition is a candidate for abstraction. As shown later in the paper, each failure condition in our model involves only a very small amount of information which allows abstraction to produce a substantial reduction in the size of the verification problem. This work extends the method in [30] to allow for abstraction independent of the hierarchical structure of the design. In other words, the method can now be applied to flat designs or designs which include large modules. This is desirable in that it eliminates the requirement of functionally unnatural partitioning for the underlying timing verification tool and the time spent in searching for such a partition. It also avoids errors incurred during decomposition. The decomposition and abstraction method described in this paper is proven to never produce a false positive verification result. Although the method can produce a false negative result, this paper describes an algorithm that examines the abstract error trace either to determine a concrete error trace or report that the result is a false negative. Finally, this paper demonstrates the effectiveness of this method by its application to two large-scale timed circuit designs.

2. Timed Petri-Nets

Our method uses *timed Petri nets* [23] to specify timed circuit behaviors. Let W be a finite set of wires in a timed circuit. The timed behavior of a circuit is modeled as sequences of rising and falling transitions on W. For any $w \in W$, $w+$ is a rising transition and $w-$ is a falling transition on the wire w. In the following definitions, let $\mathbb{Q}^+$ and $\mathbb{R}^+$ denote the sets of non-negative rational and non-negative real numbers, respectively. A W-labeled one-safe timed Petri net (TPN) is a directed bipartite digraph described by the tuple $N = (T, P, F, M_0, \mathsf{l}, \mathsf{u}, C, L)$ where T is the set of transitions; P is the set of places; $F \subseteq (T \times P) \cup (P \times T)$ is the flow relation; $M_0 \subseteq P$ is the initial marking; $\mathsf{l} : P \rightarrow \mathbb{Q}^+$ is the lower timing bound function; $\mathsf{u} : P \rightarrow \mathbb{Q}^+ \cup \{\infty\}$ is the upper timing bound function; $C \subseteq P$ is the set of *constraint places*; and $L : T \rightarrow (W \times \{+, -\})$ is the labeling function.

A transistor diagram for a self-reseting AND gate with specific timing information and a TPN representing its behavior and that of its environment are shown in Figure 1(a). A self-reseting AND gate receives a pulse on input *i1* and *i2* and generates a pulse on output a. Intuitively, the TPN shows that *i1* and *i2* go high after 11 to 14 time units. After 3 to 4 more time units, a goes high. Also, after 8 to 10 time units, *i1* and *i2* go low. The internal signal x goes low 8 to 10 time units after a goes high. This in turn resets a 1 to 2 time units later which sets x high after 1 to 2 more time units returning the circuit to its initial state.

The self-reseting AND gate is correct if it satisfies the following requirements: (1) **hold time**: the signal a must go high 1 time unit before either *i1* or *i2* goes low; (2) **short circuit**: the signal x must not go low until 1 time unit after both *i1* and *i2* have gone low, and *i1* and *i2* must not go high again until 1 time unit after x has gone high. Constraint places are used to specify these types of ordering and timing requirements between transitions. The constraint places marked with a 'C' in Figure 1(b) are used to check the above requirements. For example, the hold time requirement is checked using constraint places in the postset of $a+$.

The remainder of this section describes the formal semantics of TPNs in more detail. The state of a Petri net is a marking, M, which is the set of places that hold tokens. With every transition $t \in T$, its associated preset is $\bullet t = \{p \in P \mid (p, t) \in F\}$. The *place-set* of a transition is the restriction of places in its preset to ordinary (not constraint) places, i.e., $P(t) = \bullet t - C$. For a transition $t \in T$, its associated *postset* is $t\bullet = \{p \in P \mid (t, p) \in F\}$. Note that the preset and postset for places are defined in a similar manner. A transition is *enabled* in M if $P(t) \subseteq M$. The set of transitions enabled in M is denoted by $X(M)$. Our method requires correct nets to be *one-safe* (i.e., each place is allowed to contain no more than one token.).[1]

The state of a TPN is a pair (M, D) where M is the

[1] As described later, our analysis method checks for violations of the one-safe property during analysis, and when such a violation is detected a failure is reported and analysis ceases.

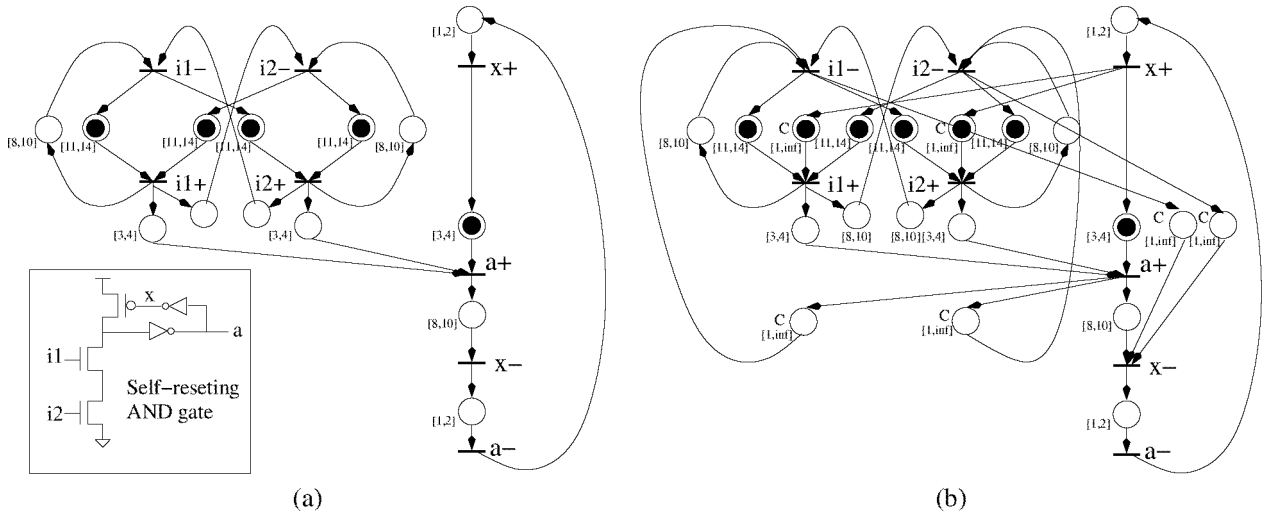

Figure 1. (a) TPN for a self-reseting AND gate. (b) TPN including timing constraints.

current marking and $D : P \rightarrow \mathbb{R}^+$ is a clock assignment function assigning nonnegative reals to places. For every place p, the value $D(p)$ is the value of a clock associated with p denoting its age. There are two operations on clocks: advance and reset. For some non-negative real number $d \in \mathbb{R}^+$, $D + d$ advances the clock for every $p \in P$ to the value $D(p) + d$. For some subset of places $\hat{P} \subseteq P$, $[\hat{P} \mapsto 0]D$ resets the clock for every place in $\hat{P}$ to zero and agrees with D for every place in $P - \hat{P}$. The initial clock assignment, D_0, is defined such that every clock is zero. The initial state of a TPN is the pair (M_0, D_0).

The state of a TPN can change by firing a transition or advancing time. To fire a transition, t, at (M, D), in addition to t being enabled, D must satisfy the timing constraints in l and u. A transition is *time-enabled* if it is enabled and: (1) the clock for each place in its place-set is above its lower bound (i.e., $\forall p \in P(t)\ .\ D(p) \geq \mathsf{l}(p)$); and (2) there exists a clock for a place in its place-set that is below its upper bound (i.e., $\exists p' \in P(t)\ .\ D(p') \leq \mathsf{u}(p')$). Firing a time-enabled transition, t, from (M, D) creates the new state (M', D') denoted by $(M, D)\,[t\rangle\,(M', D')$, where $M' = (M - {\bullet}t) \cup t{\bullet}$ and $D' = [t{\bullet} \mapsto 0]D$.

The state of a TPN can also change by advancing time. Advancing time only affects the clock assignment function in the state pair. Advancing time by a delay $d \in \mathbb{R}^+$ in (M, D) creates a new state (M, D'), denoted by $(M, D)\,[d\rangle\,(M, D')$, where $D' = D + d$. Time is not allowed to advance beyond the point where it would disable a time-enabled transition. The *maximum delay advancement*, $d \in \mathbb{R}^+$, at state (M, D) is

$$d_{\max}(M, D) = \min_{t \in X(M)} \left(\max_{p \in P(t)} (\mathsf{u}(p) - D(p)) \right).$$

After advancing time by the maximum delay, a transition either remains not time-enabled, becomes time-enabled, or is already time-enabled and remains so.

This paper uses trace theory to define semantics for TPNs. *Trace theory* has been used for the verification of both speed-independent [7] and timed circuits [29]. Given a TPN N, a *trace* of N is a sequence of transition-time pairs, (t_i, τ_i). The time, τ_i, is an absolute time stamp for t_i. The trace $((t_1, \tau_1), \ldots, (t_n, \tau_n))$ is a *valid trace* if there exists a sequence of states $(s_0, s_1, \ldots, s_n)$ such that for $1 \leq i \leq n$ each $s_i = (M_i, D_i)$ and $d_i = \tau_i - \tau_{i-1}$ (note $\tau_0 = 0$),

1. $0 \leq d_i \leq d_{\max}(s_{i-1})$;
2. $(M_{i-1}, D_{i-1})\,[d_i\rangle\,(M_{i-1}, D')$
3. t_i is time-enabled in (M_{i-1}, D'); and
4. $(M_{i-1}, D')\,[t_i\rangle\,(M_i, D_i)$

The set of all possible valid traces for a TPN N starting from the initial state (M_0, D_0) is denoted by $\mathcal{P}(N)$.

The delete function, $\mathbf{del}(\mathcal{D})(x)$, removes all transition-time pairs of a trace $x = (e_1, e_2, ...)$ whose transitions are in $\mathcal{D}$. More formally, if $x \neq \epsilon$ (i.e., the empty trace), then

$$\mathbf{del}(\mathcal{D})(x) = \begin{cases} (e_1, y) & \text{if } t_1 \notin \mathcal{D} \\ (y) & \text{if } t_1 \in \mathcal{D} \end{cases}$$

where $y = \mathbf{del}(\mathcal{D})(e_2, e_3, ...)$ and $e_i = (t_i, \tau_i)$. If $x = \epsilon$, then $\mathbf{del}(\mathcal{D})(x) = \{\epsilon\}$. This function is extended naturally to sets of traces.

The set of valid traces in a TPN is divided into those that are successes and those that are failures. There are three types of failures that are considered in this paper:

safety, *complement*, and *constraint failures*. A valid trace is a *safety failure* if in firing the trace the marking update tries to add to the new marking a place that already exists in the current marking. The one-safe requirement of TPNs is common for timed state space exploration algorithms. An unsafe net (i.e., one that is not one-safe) typically indicates a problem with the design. Note that this definition of safety is on the reachable state space, so while the TPN may not be structurally safe in an untimed sense, a failure is only reported when a marking is actually reached that violates the safety property. A valid trace is a *complement failure* on wire w if there exists two rising (falling) transitions on w without a falling (rising) transition in-between. Complement failures are also a common modeling error typically caused by the designer while creating the circuit description when the set and reset phase of a signal are similar. A valid trace is a *constraint failure* if it contains a transition or time progress that could not have occurred if constraint places are taken into account in the definition of enabledness. Constraints are used to indicate required ordering and timing relationships, and they are the key tool for describing necessary properties of a circuit such as hold time, short circuit avoidance, etc. There are three failure conditions for constraints. First, a transition having a constraint place in its preset is taken while the constraint place is not marked or has not been marked long enough. This indicates either a desired ordering of signals that is violated, or a minimum time separation between signals does not hold. The second part of the definition indicates when a token stays in a constraint place beyond its upper bound. This is used to set maximum time separations between transitions. The third part states the condition when a circuit deadlocks while a constraint place is marked. It is used to check that a desired behavior occurs before the circuit deadlocks.

In our method, the function $\mathbf{fail}(N, W', C')$ is introduced to take a TPN N, a set of wires W', and a set of constraint places C', and returns a subset of $\mathcal{P}(N)$ that are either safety failures, complement failures on W', or constraint failures involving places in C'. In other words, a valid trace $((t_1, \tau_1), \ldots, (t_n, \tau_n))$ is returned by $\mathbf{fail}(N, W', C')$ if for its corresponding state sequence $(s_0, s_1, \ldots, s_n)$ one of the following conditions is true:

1. **Safety failure**: there exist $s_{i-1} = (M_{i-1}, D_{i-1})$ and pair (t_i, τ_i) where $(M_{i-1} - {\bullet t_i}) \cap t_i{\bullet} \neq \emptyset$.

2. **Complement failure**: there exist a $w \in W'$ and pairs (t_i, τ_i) and (t_k, τ_k) such that the following is true:

   (a) $i < k$;

   (b) $(L(t_i) = L(t_k) = w+) \vee (L(t_i) = L(t_k) = w-)$;

   (c) $\forall j \,.\, i < j < k \wedge (L(t_i) = w+ \implies L(t_j) \neq w-) \wedge (L(t_i) = w- \implies L(t_j) \neq w+)$.

3. **Constraint failure**: there exist a $c \in C'$, $s_{i-1} = (M_{i-1}, D_{i-1})$, and $d_i = \tau_i - \tau_{i-1}$, such that one of the three following conditions hold:

   (a) $c \in {\bullet t_i} \wedge ((c \notin M_{i-1}) \vee (D_{i-1}(c) + d_i < \mathsf{l}(c)))$;

   (b) $c \in M_{i-1} \wedge D_{i-1}(c) + d_i > \mathsf{u}(c)$; or

   (c) $X(M_i) = \emptyset \wedge c \in M_i$.

3. Failure Directed Abstraction

A timed circuit description is defined to be *correct* if $\mathbf{fail}(N, W, C) = \emptyset$. This section presents an approach to proving $\mathbf{fail}(N, W, C) = \emptyset$ by showing that:

1. $\mathbf{fail}(N, \emptyset, \emptyset) = \emptyset$,

2. $\forall w \in W \,.\, \mathbf{fail}(N, \{w\}, \emptyset) = \emptyset$, and

3. $\forall c \in C \,.\, \mathbf{fail}(N, \emptyset, \{c\}) = \emptyset$.

Now, instead of one verification run, our method performs $1 + |W| + |C|$ runs. Note that $\mathbf{fail}(N, \emptyset, \emptyset)$ checks safety properties explicitly, but when $|W| + |C| \geq 1$, this does not need to be done as a separate step since it is checked implicitly during the other checks.

At this point, each run is nearly as complex as the original run, but for each subproblem, not all transitions in N are required to determine if failure traces exist. Therefore, in the second step, our method constructs a set of transitions that can be safely abstracted based on the given failure definition. The function $\mathcal{D}(N, W', C')$ takes a set of wires ($W' \subseteq W$) and a set of constraint places ($C' \subseteq C$), and it returns the following set:

$$\{t \in T \mid (\forall w \in W' \,.\, L(t) \neq w+ \;\wedge\; L(t) \neq w-) \;\wedge\; (\forall c \in C' \,.\, t \notin {\bullet c} \cup c{\bullet})\}$$

Finally, the third step of our method is to apply *safe transformations* to the net to remove these transitions and the related places, whenever possible. A transformation, $\pi_i(N)$, returns a new net N', and it is defined to be safe when the TPN resulting from this transformation satisfies the following two properties:

$$\begin{aligned} \mathcal{P}(N') &\supseteq \mathbf{del}(T - T')(\mathcal{P}(N)) \\ \mathbf{fail}(N', \emptyset, \emptyset) &\supseteq \mathbf{del}(T - T')(\mathbf{fail}(N, \emptyset, \emptyset)) \end{aligned}$$

where T' is the set of transitions in N'. In other words, a net produced by a safe transformation produces a superset of the timed traces produced by the original TPN when any *abstracted transitions* are deleted from these traces, and the transformation does not hide a safety failure of the net. As shown in the following lemma, the application of a sequence of safe transformations is also a safe transformation.

Lemma 3.1 *If $\pi_i(N)$ and $\pi_j(N)$ are safe transformations, then so is $\pi_j(\pi_i(N))$.*

Proof: Assume $N' = \pi_i(N)$ and $N'' = \pi_j(N')$. From the definition of safe transformation, we have:

$$\begin{aligned} \mathcal{P}(N') &\supseteq \mathbf{del}(T - T')(\mathcal{P}(N)) \\ \mathcal{P}(N'') &\supseteq \mathbf{del}(T' - T'')(\mathcal{P}(N')) \end{aligned}$$

Combining these two equations, we get:

$$\begin{aligned} \mathcal{P}(N'') &\supseteq \mathbf{del}(T' - T'')(\mathbf{del}(T - T')(\mathcal{P}(N))) \\ &= \mathbf{del}(T - T'')(\mathcal{P}(N)) \end{aligned}$$

This proves the first half of the definition of safe transformation. The second half (i.e., $\mathbf{fail}(N'', \emptyset, \emptyset) \supseteq \mathbf{del}(T - T'')(\mathbf{fail}(N, \emptyset, \emptyset))$) is proven similarly. ∎

We define a function $\mathbf{abs}(N, W'', C'')$ that takes a TPN N, a set of wires W'', and a set of constraint places C'', and applies a sequence of safe transformations to remove, when possible, transitions in $\mathcal{D}(N, W'', C'')$ from N to obtain a new TPN N'. The safe transformations used are restricted such that $T - T' \subseteq \mathcal{D}(N, W'', C'')$ and for all $c \in C''$, c is in the initial marking of the new net, M_0', if and only if c is in initial marking of the original net, M_0. The result after applying this function to a net is typically a net that is substantially simpler and thus results in a much smaller state space. The main theorem can now be presented.

Theorem 3.1 *Let N be A TPN.* $\mathbf{fail}(N, W, C) = \emptyset$ *if the following three conditions are true:*

1. $\mathbf{fail}(\mathbf{abs}(N, \emptyset, \emptyset), \emptyset, \emptyset) = \emptyset$.
2. $\forall w \in W$. $\mathbf{fail}(\mathbf{abs}(N, \{w\}, \emptyset), \{w\}, \emptyset) = \emptyset$.
3. $\forall c \in C$. $\mathbf{fail}(\mathbf{abs}(N, \emptyset, \{c\}), \emptyset, \{c\}) = \emptyset$.

Proof: We break up this proof into three cases.

Case 1: (safety failures) Assume there is a trace x that causes a safety failure in N, and that N' is the TPN returned by the function $\mathbf{abs}(N, \emptyset, \emptyset)$. Since $x \in \mathbf{fail}(N, \emptyset, \emptyset)$, there must also exist a trace $y \in \mathbf{del}(T - T')(\mathbf{fail}(N, \emptyset, \emptyset))$. We know that y must also be in $\mathbf{fail}(N', \emptyset, \emptyset)$ since safe transformations are required not to hide safety violations. Therefore, a safety failure is detected on the abstracted net.

Case 2: (complement failures) Assume there is a trace x that causes a complement failure on signal w in N, and that N' is the TPN returned by the function $\mathbf{abs}(N, \{w\}, \emptyset)$. Since $x \in \mathcal{P}(N)$, there must also exist a trace $y \in \mathbf{del}(T - T')(\mathcal{P}(N))$. We know that y must also be in $\mathcal{P}(N')$ since safe transformations do not hide any timed traces. From the definition of complement failure, there exists two transitions t_i and t_k on signal w that create the complement failure. In fact, only transitions on w are required to show if a trace is or is not a complement failure. By the definition of **abs**, the trace y must include all transitions on signal w in trace x with some additional transitions from $\mathcal{D}(N, \{w\}, \emptyset)$ which could not be abstracted. Since a complement failure is detected by only examining those transitions on signal w and x is a complement failure, y is also a complement failure because removing transitions not on w does not change whether a trace is a complement failure or not.

Case 3: (constraint failures) Assume there is a trace x that causes a constraint failure on constraint place c in N, and that N' is the TPN returned by the function $\mathbf{abs}(N, \{w\}, \emptyset)$. Since $x \in \mathcal{P}(N)$, there must also exist a trace $y \in \mathbf{del}(T - T')(\mathcal{P}(N))$. This trace consists of all transitions in $\bullet c \cup c\bullet$ plus some additional transitions from $\mathcal{D}(N, \emptyset, \{c\})$ which could not be abstracted. Traces x and y agree on the timing of all transitions in $\bullet c \cup c\bullet$. There are three reasons that trace x is a constraint failure (cases 3(a), 3(b), and 3(c) shown above). For case 3(a), since all transitions in $\bullet c$ are preserved as is the initial marking of c, the value of the predicate $c \notin M_{i-1}$ is preserved at the time of firing t_i (a transition which is also preserved). The value of $D_{i-1}(c)$ is also preserved for this reason. Therefore, if trace x violates 3(a), so does trace y. If trace x violates case 3(b), this means a transition from $\bullet c$ fired to put a token into c, then either a transition in $c\bullet$ fired but fired too late (in this case it is preserved in y, so it is okay) or a transition outside $\bullet c \cup c\bullet$ fired to cause the failure. In this case, that transition is not necessarily preserved in y. However, either there exists another transition in y that causes the upper bound violation, or if the trace is finite and ends with no transitions being enabled, y is a failure due to case 3(c). In either case, the failure is found examining y. Finally, case 3(c) is preserved from x to y as in both the trace ends with no enabled transitions and the constraint place in the marking.

Therefore, if there exists a failure trace in the concrete description, it is found by an analysis of one of the abstract descriptions. ∎

In the rest of this section, we discuss the safe transformations that can be used in the third step of the above method. Murata and others [21] present several transformations on untimed Petri nets that preserve the safety properties of the original net. We have extended these transformations and developed others for TPNs [30]. Two example safe transformations are shown in Figures 2 and 3. If our method is working on a net N and finds a portion of the net that resembles that shown in Figure 2(a) and t can be abstracted, it can transform it to a new net N' in which t has been removed as shown in Figure 2(b) in which the timing bounds have been combined as shown to preserve the timing behavior. Note that although shown with only two places in the preset of t, this transformation is valid for any number of places in the preset of t as long as there is only one place in the postset of t. While the places in the preset of t can have any number of transitions in their presets, they must only have transition t in their postset (i.e., $(\bullet t)\bullet = \{t\}$). Similarly, the place

in the postset of t can have any number of transitions in its postset, but it must only have transition t in its preset (i.e., $\bullet(t\bullet) = \{t\}$). In a similar fashion, if transition t has only a single place in its preset and satisfies similar restrictions, it can again be removed as shown in Figure 3. The application of these transformations is quite efficient. Most have complexity that is linear in the size of the net while some are quadratic.

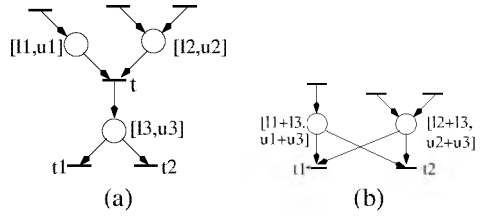

Figure 2. Safe transformation 1.

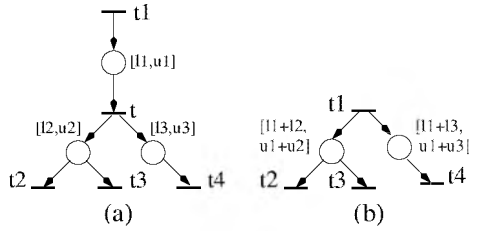

Figure 3. Safe transformation 2.

The TPN to check for safety failures in our self-reseting AND gate example after applying safe transformations is shown in Figure 4. A timing analysis of this net shows that $i2+$ fires after 19 to 24 time units, followed by $a+$ after 3 to 4 time units which enables $i2+$ to fire again after another 19 to 24 time units. Notice that the self-loop on $a+$ is never constraining. This shows how close the original TPN is to having a safety failure. If the maximum delay on any transition on signal a or x is increased by one time unit, there is a safety failure. Note that the complement failures on signals $i2$ and a are ignored during the safety failure check verification run. The decomposition method described in this paper would verify this example using 11 verification runs which is clearly overkill for such a small example. For large examples such as the RAPPID example described later, the substantial net reductions possible are quite beneficial to improving overall verification time.

The verification method just described is conservative in that false negatives are possible though false positives never result. Consider again the transformation shown in Figure 3. In this case, the summing of the timing bounds as shown in the figure may actually result in new timed traces. For example, in the new net, the trace $((t_1, 0), (t_2, l_1 + l_2), (t_4, u_1 + u_3))$ is possible, while this is not possible in the original net. It does, however, produce all the timed traces of the original net, so it is a safe transformation. If this extra timing introduces new timed traces and one of those introduced traces causes a failure, this failure is a false negative. Therefore, when an error trace is reported from an analysis of the abstracted net, it is not known whether this is a real error trace or a false one. Also, it is difficult for a designer to analyze the error trace to find the problem as it only includes the transitions which have not been abstracted.

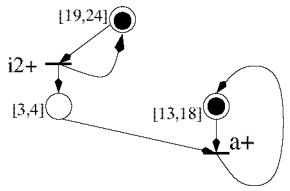

Figure 4. TPN for checking safety.

To address both of these problems, this paper introduces the algorithm shown in Figure 5. This algorithm uses the abstract error trace to perform a guided simulation of the original TPN to find a concrete error trace. This is done by attempting to fire transitions from the abstract error trace, and when one of these transitions is not fireable, it examines the TPN to determine an abstracted transition to fire that contributes toward the enabling of the next transition in the abstract error trace. In general, multiple such transitions may exist and the algorithm may need to explore multiple paths to find a valid concrete error trace. When no concrete error trace can be found, it is reported that the abstract error trace is false. In this case, abstraction is performed again without using transformations such as the one shown in Figure 3 that are known to add timed traces. While in the worst-case this can result in a flat verification, we have not seen this happen in practice.

4. Experimental Results

We have incorporated the method described in this paper into the compiler front-end of the ATACS tool [22]. This tool also includes the modular verification method from [30]. In the following experiments, the flat, modular, and failure directed approaches use the same explicit-state reachability analysis engine and parameter settings [18]. This section describes the application of our method to two large examples, and the results are shown in Table 1.

The first is Intel's RAPPID circuit which is a fully asynchronous instruction-length decoder for the PentiumII 32-bit MMX instruction set [24]. In this instruction set, each instruction can be from 1 to 15 bytes long, depending on a large number of factors. In order to allow concurrent ex-

Table 1. Experimental results.

	Flat			Modular			Failure Directed		
	States (max)	Mem (Mb)	Time (min)	States (max)	Mem (Mb)	Time (min)	States (max)	Mem (Mb)	Time (min)
RAPPID	>160,000	>576	>130	20,120	76	6.2	234	11	2.2
TITAC2	>226,000	>576	>360	n/a	n/a	n/a	120	30	23

```
find_concrete_trace(N, D, x)
  s = (M_0, D_0)
  (t, τ) = head(x)
  x = tail(x)
  x' = ε
  do
    if ((t, τ) is time-enabled in s) then
      (t', τ') = (t, τ)
      (t, τ) = head(x)
      x = tail(x)
    else
      E = necessary_set(t, s) ∩ D
      if (E = ∅) then
        if (stack not empty) then
          pop(s, x, x', E)
        else return false negative
      (t', τ') = choose_one(E)
      E = E − {(t', τ')}
      if (E ≠ ∅) then push(s, x, x', E)
    s = fire((t', τ'), s)
    x' = (x', (t', τ'))
  while (x is not empty)
  return x'
```

Figure 5. Algorithm to find concrete trace.

ecution of instructions, it is necessary to rapidly determine the positions of each instruction in a cache line. Instruction-length decoding was a critical performance bottleneck in the PentiumII architecture at the time when RAPPID was being designed. The RAPPID circuit is shown to perform three times faster while using half the power of the comparable synchronous design. This performance improvement is due in large part to the highly timed nature of the circuits in this design. Therefore, the correctness of this design is highly dependent on timing parameters. The block diagram for the portion of the RAPPID design that we verified is depicted in Figure 6. The TPN description of the RAPPID circuit has 115 transitions on 49 signal wires. Flat analysis runs out of memory after two hours on a 650 MHz PentiumIII with 576 megabytes of memory (the stack depth was in excess of 33,000 entries and climbing indicating that it had a long way to go). The modular approach [30] decomposes the verification problem into 10 subproblems, one for each module shown in Figure 6. The total verification time is 6.2 minutes for all 10 verification runs. The largest module is IR which takes 5.2 minutes to explore 20,120 timed states using 76 MB of memory. Our failure model directed approach presented in this paper decomposes the verification problem into 50 subproblems to check for safety, complement, and constraint failures. The total verification time is 2.2 minutes for all 50 verification runs. The largest verification run is for checking complement failures on the signal TagOut4 which takes 3.2 seconds to explore 234 timed states using 11 MB of memory.

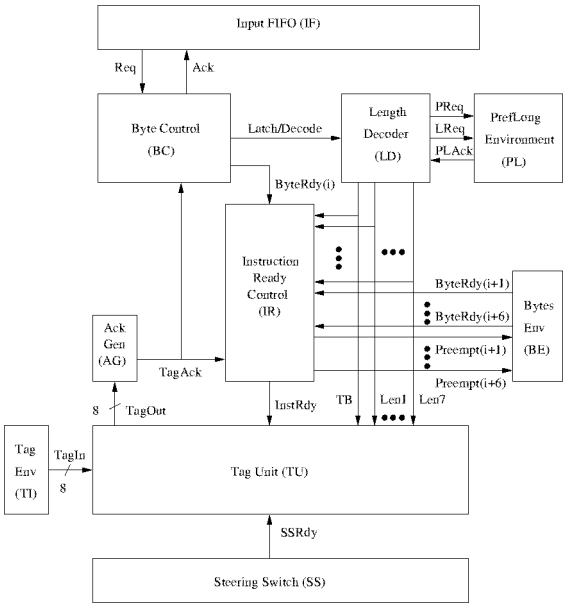

Figure 6. Block diagram for RAPPID circuit.

Our second example is the line fetch module of TITAC2's instruction cache system [25] which is represented using a TPN with 156 transitions derived from a high-level specification [28]. Verification of this one module did not complete after running for 4 hours after exploring more than 226,000 timed states. This example is a single flat module, so the modular approach cannot be used. The failure model approach decomposes the verification of this module into 62 verification runs each with less than 120 timed states with a total verification time of 23 minutes.

5. Conclusions and Future Work

This paper describes a new method to deal with state explosion by decomposing the timing verification problem as directed by the given failure model. This decomposition allows for a significant reduction in the size of the model for each subproblem using an automatic abstraction method based on safe transformations. It no longer requires that a design is properly partitioned for successful verification. This method has been applied to two large timed circuit designs including one that could not previously be verified. Overall, this method scales very well in that the size of the individual verification problems are only dependent on the complexity associated with a single signal or a single constraint place. This new method can also be built on top of any reachability analysis algorithm for timed Petri nets, and benefit from any improvements in the underlying analysis algorithm. In particular, our preliminary analysis has shown that combining abstraction with a partial order based analysis technique can bring even further improvements.

References

[1] W. Belluomini and C. J. Myers. Timed circuit verification using tel structures. *IEEE Transactions on Computer-Aided Design*, 20(1):129–146, Jan. 2001.

[2] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed systems. In *International Conference on Concurrency Theory*, pages 485–500, 1998.

[3] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Proc. Int. Conf on Computer Aided Verification*, 1997.

[4] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided-Design*, pages 401–424, April 1994.

[5] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[6] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[7] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[8] P. Godefroid. Using partial orders to improve automatic verification methods. In *International Conference on Computer-Aided Verification*, pages 176–185, June 1990.

[9] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16:843–872, 1994.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *The 29th Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.

[11] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. Int. Conf on Computer Aided Verification*, pages 440–451. Springer-Verlag, 1998.

[12] H. P. Hofstee, S. H. Dhong, D. Meltzer, K. J. Nowka, J. A. Silberman, J. L. Burns, S. D. Posluszny, and O. Takahashi. Designing for a gigahertz. *IEEE MICRO*, May-June 1998.

[13] H. E. Jensen, K. G. Larsen, and A. Skou. Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT*, pages 19–30, 2000.

[14] C. Jones. Tentative steps toward a development for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[15] K. Larsen, B. Steffen, and C. Weise. A constraint oriented proof methodology. In *Formal Systems Verification*, volume 1169 of *LNCS*, pages 405–435. Springer-Verlag, Nov. 1996.

[16] K. G. Larsen, C. Weise, Y. Wang, and J. Pearson. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.

[17] K. L. Mcmillan. A methodology for hardware verification using compositional model checking. Technical report, Cadence Berkeley Labs, 1999.

[18] E. Mercer, C. Myers, and T. Yoneda. Improved poset timing analysis in timed petri nets. In *The 10th Workshop on Synthesis and System Integration of MIxed Tech.*, Oct. 2001.

[19] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. on Software Eng.*, SE-7(4):417–426, 1981.

[20] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, The IT University of Copenhagen, Denmark, Sept. 1999.

[21] T. Murata. Petri nets: Properties, analysis, and applications. In *Proceedings of the IEEE 77(4)*, pages 541–580, 1989.

[22] C. J. Myers, W. Belluomini, K. Killpack, E. Mercer, E. Peskin, and H. Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, Feb. 2001.

[23] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Tech. Rep. 120, MIT, Feb. 1974.

[24] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, Feb. 2001.

[25] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model. In *Proc. International Conf Computer Design (ICCD)*, pages 288–294, Oct. 1997.

[26] S. Tasiran and R. K. Brayton. Stari: A case study in compositional and heirarchical timing verification. In *Proc. Int. Conf on Computer Aided Verification*, 1997.

[27] A. Valmari. A stubborn attack on state explosion. In *International Conference on Computer-Aided Verification*, pages 176–185, June 1990.

[28] T. Yoneda and C. Myers. Synthesizing timed circuits from high level specification languages. *NII Technical Report*, NII-2003-003E, 2003.

[29] T. Yoneda and H. Ryu. Timed trace theoretic verification using partial order reduction. In *Proc. Int. Sym. on Asynchronous Circuits and Systems*, pages 108–121, Apr. 1999.

[30] H. Zheng, E. Mercer, and C. J. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Transactions on Computer-Aided Design*, 22(9), Sept. 2003.